AFCRL-67-0167

CRT-AIDED SEMI-AUTOMATED MATHEMATICS

James H. Bennett
William B. Easton
James R. Guard
Larry G. Settle

APPLIED LOGIC CORPORATION
ONE PALMER SQUARE
PRINCETON, NEW JERSEY

Contract AF 19(628)-3250
Projec No. 8672

FINAL REPORT

Period Covered: 1 June 1963 through 30 September 1966

January 1967

This research was sponsored by the Advanced Research Projects Agency under ARPA Order No. 700

Contract Monitor: Timothy P. Hart

Distribution of this document is unlimited

Prepared
for

AIR FORCE CAMBRIDGE RESEARCH LABORATORIES
OFFICE OF AEROSPACE RESEARCH
UNITED STATES AIR FORCE
BEDFORD, MASSACHUSETTS 01730

60

AFCRL-67-0167

CRT-AIDED SEMI-AUTOMATED MATHEMATICS

James H. Bennett
William B. Easton
James R. Guard
Larry G. Settle

APPLIED LOGIC CORPORATION
ONE PALMER SQUARE
PRINCETON, NEW JERSEY

Contract AF 19(628)-3250
Project No. 8672

FINAL REPORT

Period Covered: 1 June 1963 through 30 September 1966

January 1967

This research was sponsored by the Advanced Research Projects Agency under ARPA Order No. 700

Contract Monitor: Timothy P. Hart

Distribution of this document is unlimited

Prepared
for

AIR FORCE CAMBRIDGE RESEARCH LABORATORIES
OFFICE OF AEROSPACE RESEARCH
UNITED STATES AIR FORCE
BEDFORD, MASSACHUSETTS 01730

TABLE OF CONTENTS

ACKNOWLEDGMENTS

The authors wish to acknowledge the encouragement given them during every phase of the work by their technical monitor, Timothy Hart, and by Ivan E. Sutherland of the Advanced Research Projects Agency.

During the early phases of the work three of the authors were affiliated with Princeton University, and one of them was also affiliated with The University of Michigan. All wish to acknowledge the encouragement and support they have received from their respective universities.

Finally, thanks to the staffs of Applied Logic Corporation, Socony Mobil Oil Co. Inc., and the Princeton and Rutgers University Computation Centers for their helpful assistance.

PERSONNEL

The authors wish to express appreciation for the whole-hearted efforts on the part of the following personnel:

Roger Haydock
David B. Loveman
Thomas H. Mott, Jr.
Francis C. Oglesby
William L. Paschke
Eleanore G. Wells

ABSTRACT

This report describes the status of the fifth in a series of five experiments in semi-automated mathematics. This effort extended from June 1, 1963 through September 30, 1966. These experiments culminated in large complex computer programs which allow a mathematician to prove mathematical theorems on a man-machine basis. SAM V, the fifth program, uses a cathode ray tube as the principal interface between the mathematician and a high speed digital computer. An elaborate language and logical capability has been implemented in SAM V. These include I/O languages for expressing mathematical statements in a form suitable for both the mathematician and the machine to recognize and handle with ease and convenience; a language for expressing and handling sorts and range of symbols; and an auto-logic algorithm and matching routine. The latter constitute the capability for handling, automatically, logic with equality. This capability is particularly useful at an intermediate state of the proof when it is desired to have the machine try and verify automatically a given portion of the proof.

SUMMARY

This final report describes a series of five computer programs, called SAM I through V, which are experimental tools for studying techniques in theorem proving via human interaction with computers. The approach of semi-automated mathematics which underlies this series of programs is that of using man-machine interaction to achieve results which neither component could achieve alone. The first four programs are described in detail in [1, 2, 3, 4] (See Bibliography). Each of the five programs concentrated on attacking specific phases of the problem. The current program is oriented primarily toward the development of efficient automatic techniques for handling some of the smallest processes of mathematical deduction and toward the realization of efficient real time interaction between man and machine through the use of CRT displays.

The first program, SAM I, implemented the propositional calculus in a framework of natural deduction; the goal of human intervention in SAM I was to obtain proofs of minimal length. SAM II dealt with quantifier-free first-order axiom systems of mathematics. SAM II was adequate to investigate elementary mathematical theories including geometry and elementary set theory. The program left the entire burden of proof generation with the user. SAM II was responsible for checking the validity of steps and generating consequences by the basic rules. SAM III saw the beginning of the development of auto-logic, which contained the capability for automatically handling predicate and functional logic containing equality. This capability is particularly useful at an intermediate stage of a proof

when it is desired to have the machine attempt to verify a portion of a proof without requiring the user to supply all the elementary steps in the derivation. The years have seen continual increase in the power of auto-logic to automatically verify the truth of complex deductions. SAM III initiated development of sophisticated input/output techniques and contained the first general purpose languages for expressing mathematically statements in suitable form for both mathematician and machine.

The programs, SAM I, II, and III, were implemented on a small scientific computer, the IBM 1620. SAM IV expanded the capability of SAM III in a number of directions and was implemented on an IBM 7040, a medium scale scientific computer. The improvements were primarily in auto-logic and in the use of SLIP (a list processing language) as the underlying framework for the program.

SAM V saw advances in auto-logic with respect to the semi-automatic handling of equality and the algebraic aspects of mathematical theories. It has also seen the implementation of a CRT display as the primary interface between man and machine. This is a most convenient and flexible means of interaction and the first allowing truly real time communication between man and machine at a rate that is efficient for the user. The program was implemented on a PDP-6, a large-scale computer with a time-sharing system. Time sharing is a mode of operation which allows efficient and economical interaction between man and machine at the convenience of the user.

This report expands and brings up to date the material contained in [7,8]. Our intention is to make this report a self-contained description of SAM V as it existed on September 30, 1966.

SECTION I

SYMBOLS AND FORMULAS

In this section we describe the symbols and formulas of SAM V from the viewpoint of logical structure. The symbols and formulas are the language of SAM, the fundamental entities with which the user is concerned. In semi-automatic mathematics they bear the entire responsibility for expressing theorems and steps in proofs. Mathematical investigations in symbolic logic have shown that a small collection of basic kinds of entities and rules for combining them into formulas are sufficient to represent mathematical theorems and proofs. In part, the success of SAM depends on the ability of its formulas to conveniently express mathematical ideas in a way which lends itself to efficient algorithmic methods. In Section III (Control, Input/Output) we describe the manner in which formulas and proofs are actually presented to the user in ways which promote understanding and rapid communication. In this section our description is cast in terms of representations internal to SAM. This is convenient in making precise the sense of the attributes carried by the language of SAM.

There are four types of symbols represented in SAM V. These are variables, constants, logical symbols, and punctuation. Variables and constants are represented internally by a number which corresponds to a single alphabetic letter of a single alphabetic letter with a subscript. Certain bits in this representation of a symbol indicate syntactical status of that symbol. For convenience in debugging they appear as bits reflected in the leading digits in the "subscript".

Subscripts are positive integers less than 2^7. These subscripts are written in octal notation. There is a table used in SAM V which specifies which of the symbols are constants and which are variables. The standard table has the symbols starting with A through H and P through S as constants and the remaining symbols as variables. A given variable can have three distinct representations in SAM V according as the variable is free, bound, or temporarily fixed. (A fixed variable corresponds to a variable which, in an intuitive sense, has been fixed by a statement such as "let x be a positive number"). Constants have a single internal representation. The internal representation of variables and constants is as follows: Bits 3 and 4 are 00, 11, 10, or 01 according as the symbol is a constant, variable, bound variable, or a fixed variable; bits 5 through 11 are the subscript (no subscript is represented by zero; and bits 12 through 17 are the 6-bit ASCII code for the alphabetic character. A subscript 100 through 177 indicates a "shadowed" variable. The use of "shadowed" variables is a technique used to avoid clashes of free variables. This technique is described in more detail in Section II. A subscript between 40 and 77 and 140 and 177 indicates a variable which has temporarily been changed to a constant, called a "frozen" variable. "Frozen" variables are a technical device used to simplify the matching and instantiation routines. (See the description of Matching in Section II) A subscript of 20 to 37 or 60 to 77 indicates a variable which has been turned into a constant by the Skolemizing process described below. These constants are called "Skolemized variables". The logical symbols are typed as LAM, ALL, IS, =, IMP, OR, AND, IFF, NOT, TRU, and FAL . These are represented internally respectively by the octal numbers 1 through 13 right justified in bits 3 through 17.

The punctuation symbols are left and right parentheses, comma, and left and right square brackets (the square brackets are represented by angle brackets on output). Punctuation has no internal representation and is used only for rudimentary I/O. (The rudimentary I/O language in which we express formulas in this section is convenient for debugging and explanation of Section III.)

The symbols above are combined by the following rules to form formulas:

1. A single variable or constant is a formula.
2. If b is a variable or constant and $a_1, a_2, \ldots, a_n$ are formulas, then

   $b(a_1, \ldots, a_n)$ is a formula
3. If b is a formula with more than one symbol and $a_1, \ldots, a_n$ are formulas, then

   $[b]\, a_1, \ldots, a_n$ is a formula
4. If b and c are formulas and $d_1, d_2, \ldots, d_r$ are variables, then

   (b=c)

   (ALL d_1)(b)

   (IS d_1)(b)

   (LAM $d_1, d_2, \ldots, d_r$)(b)

   (b IMP c)

   (b OR c)

   (b AND c)

   (b IFF c)

   NOT(b)

are formulas.

Formulas are represented internally by list structures. These list structures are manipulated by a package of subroutines which we have called TROLL for Threaded, Ringed, Oriented List Language. Figures 1 through 3 on the next page show the representation of two formulas as TROLL lists. A description of TROLL is included in Section V.

In SAM V, only certain types of formulas are considered internally. These formulas are those which can be formed by using variables, constants, =, LAM and NOT. This set of symbols, however, is sufficient to represent any mathematica entities which can be described in an omega-order predicate calculus.

To see this we define the miniscope form and the Skolemization of a formula. The Skolemization of a formula is logically equivalent to the original formula.

Definition. A wff is converted to its miniscope form by applying the following replacement rules (repeatedly using the first applicable rule; we write $x \notin A$ to mean x is not free in A):

Rule	Replace Subformula	by Subformula	if
1	NOT NOT A	A	
2	A IMP B	NOT A OR B	
3	A IFF B	(NOT A OR B) AND (A OR NOT B)	
4	NOT (A OR B)	NOT A AND NOT B	
5	NOT (A AND B)	NOT A OR NOT B	
6	NOT (ALL x) A	(IS x) NOT A	
7	NOT (IS x) A	(ALL x) NOT A	
8	(ALL x) A	A	$x \notin A$
9	(IS x) A	A	$x \notin A$
10	(ALL x) (A OR B)	A OR (ALL x) B	$x \notin A$
11	(ALL x) (A OR B)	(ALL x) A OR B	$x \notin A$
12	(IS x) (A AND B)	A AND (IS x) B	$x \notin A$
13	(IS x) (A AND B)	(IS x) A AND B	$x \notin A$

Rule	Replace Subformula	by Subformula
14	(ALL x) (A AND B)	(ALL x) A AND (ALL x) B
15	(IS x) (A OR B)	(IS x) A OR (IS x) B
16	(ALL x) ((A AND B) OR C)	(ALL x) (A OR C) AND (ALL x) (B OR C)
17	(ALL x) ((A OR (B AND C))	(ALL x) (A OR B) AND (ALL x) (A OR C)
18	(IS x) ((A OR B) AND C)	(IS x) (A AND C) OR (IS x) (B AND C)
19	(IS x) (A AND (B OR C))	(IS x) (A AND B) OR (IS x) (A AND C)

Definition. The Skolemization of a wff A is obtained by applying the following steps to A :

1. Completely lambda-convert A .
2. Take the universal closure of A .
3. Convert A to its miniscope form.
4. Reletter the universally bound variables so that no variable appears with two universal quantifiers.
5. Replace an occurrence of a subformula of the form (IS x) B by

$$S^{x}_{F(x_1,\ldots,x_n)} B$$

where this occurrence of (IS x) B is within the scope of universal quantifiers binding $x_1,\ldots,x_n$ $(n \geq 0)$ and F is a new constant.

6. Delete the universal quantifiers (after all possible applications of step 5 have been completed).
7. Put the result in conjunctive normal form, i.e., replace subformulas of the form (A AND B) OR C and A OR (B AND C) by (A OR C) AND (B OR C) and (A OR B) AND (A OR C) respectively.

Remark: Any quantifiers or logical connectives which occur within the scope of a non-logical function or predicate are to be ignored by the miniscope and Skolemization procedures.

In proofs, the conjuncts of the Skolemization of a formula are treated as separate formulas. The disjuncts of each of the conjuncts are represented as a list. This list is called a pseudo-disjunction (PSD). These disjuncts are either atomic formulas or negations of atomic formulas. As an example we apply Skolemization to the formula below and write it in the form used by SAM V. For example, the formula

(ALL X)(E(X U) IFF E(X, V,) IMP U=V

which says (when reading E(A, B) as $A \in B$) that two sets are equal if they have the same members, is transformed to the two PSD's:

NOT E(F(U, V), U)
NOT E(F(U, V), V)
U=V

and

E(F(U, V), U)
E(F(U, V), V)
U=V

The meaning of the new constant function F is that F(U, V) is to be a member of exactly one of the sets U or V if they are not equal. Hence if F(U, V) is in both U and V or if F(U, V) is in neither U nor V, then in fact U=V. While in this example the two PSD's generated seem somewhat remote from their progenitor, this remoteness is an exception rather than the rule. For example, the Skolemization is frequently a rather natural restatement of the original formula.

For SAM to deal effectively with any complicated mathematical structure, it must have an efficient means of distinguishing and ordering the various classes ("sorts") of variables and constants which it encounters therein. This problem does not arise in, say, a three-axiom elementary treatment of group theory, because all variables and constants are assumed to be elements of the group in question. Suppose, however, that we wish to axiomatize a system involving several distinct vector spaces over a given field of scalars. Here, some variables will stand for spaces, others for subspaces of a given space, and still others for field elements. Moreover, such constants as the zero vectors in the spaces and the two identity elements of the field must be distinguished and placed under the proper headings.

Heretofore, we have gotten around this problem in a rather artificial manner by extensive use of PSD's. In order to tell SAM that scalar multiplication distributes over vector addition, one needed to write something like:

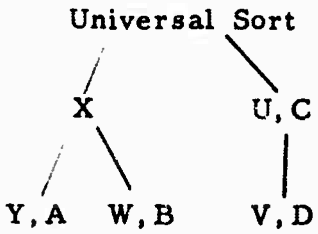
```
NOT P(Z)
NOT Q(X1, Z)
NOT Q(X2, Z)
NOT Q1(U)
G(U,H(X1,X2)) = H(G(U,X1),G(U,X2))
```

where G is scalar multiplication, H is vector addition, Q is membership in a vector space, Q1 is a membership in the scalar field, and P(Z) is the statement "Z is a vector space". Disjuncts of this kind are cumbersome on both sides of the man-machine interface.

The recent implementation of sorts in the SAM package gives us a more convenient and powerful method of handling axiom systems involving different types of variables and constants. By setting up a sort structure for the linear algebraic system mentioned above, we can indicate with a single statement that whenever an X (with or without subscript) appears in our axiomatization, it is to be considered as a member of the sort of all vectors, that subscripted or unsubscripted Y's belong to the sort of vectors in the vector space $\mathcal{Y}$, that subscripted or unsubscripted U's belong to the sort of scalars, and so on, with all constants being similarly placed in the sort to which they belong. With this set-up, it is possible to express many algebraic notions in a more straightforward manner, without the use of disjuncts. Only the equality

$$G(U, H(X1, X2)) = H(G(U, X1), G(U, X2))$$

is needed for distributivity, since the sort structure automatically sees to it that the variables U, X1, X2 are properly identified. The one major criterion which a sort structure must meet in order for SAM to be able to work with it is that the sorts be partially ordered by inclusion. That is, if two subsorts of a given sort have non-empty intersection, one of them must wholly contain the other. Admissible sort structures may thus be represented by tree diagrams such as the following for a system consisting of two vector spaces $\mathcal{Y}$ and $\mathcal{W}$ over the complex numbers:



Here, X represents the sort of all vectors, Y represents the sort of vectors in the space $\mathcal{Y}$, W the sort of vectors in $\mathcal{W}$, U the sort of complex numbers, and V the sort of reals. The diagram also establishes the convention that distinguished vectors in $\mathcal{Y}$ (the zero vector, basis vectors, etc.) will be denoted by subscripted or unsubscripted A's and similarly for B, C, and D . We are thus spared the necessity of using a separate axiom to place each distinguished constant in the sort to which it belongs. The above sort structure is entered very economically in SAM with the statement

$$S(S(X, S(Y, A), S(W, B)), S(U, C, S(V, D)))$$

which is inserted separately from the axioms for the system.

SECTION II

AUTO-LOGIC

The purpose of the AUTO-LOGIC routine is to generate "interesting" consequences of a finite set of pseudo disjunctions. Such a routine is useful in two ways: firstly, it can be used to generate new theorems which may be of interest to the mathematician and which may be useful in further applications of AUTO-LOGIC; secondly, a formula A is proved to be a consequence of a list of pseudo-disjunctions if FAL is obtained as a consequence of the list augmented by the PSD or PSD's representing the logical negation of A. The underlying principle by which AUTO-LOGIC generates useful consequences is as follows. It has four processes called reduction, expansion, digression, and contradiction. Reduction uses a set of PSD's from the initial set to 'reduce' or 'simplify' a given PSD from the set using the logical rules of an omega-order predicate-function calculus with equality and lambda notation. Expansion and digression use these same rules to generate new PSD's from a finite set of PSD's. Contradiction eliminates "trivial" PSD's by automatically Skolemizing a copy of the negation of the PSD and attempting to find a contradiction in a limited period of computer time. AUTO-LOGIC starts with a finite set of PSD's and applies these four processes in a pattern which allows the newly generated PSD's to stay in the set only if they cannot be reduced by reduction or deleted by contradiction.

SAM V lays special emphasis on developing and experimenting with different reduction, expansion, digression, and contradiction processes as well as various patterns for applying these processes.

The reductions which are currently used in AUTO-LOGIC are of two types.

The first type, called self-reduction, reduces a single PSD. Self-reduction makes the following obvious kinds of simplification:

1. The PSD is deleted if it contains a disjunct of the form b=b or a disjunct of the form TRU (in this and similar cases we say the PSD has been reduced to TRU).
2. If the PSD contains a disjunct of the form NOT(b=b) or a disjunct of the form FAL then such disjuncts are deleted. If there were no additional disjuncts we say that the original PSD disjunction was reduced to FAL . In this latter case the main control of AUTO-LOGIC is notified that a contradiction has been found.
3. If two disjuncts occur, one of which is the negation of the other, the PSD is reduced to TRU and the PSD is deleted.
4. If two identical disjuncts occur, one is deleted.
5. A disjunct of the form NOT(NOT(b)) is replaced by b .

The second type of reduction uses a single PSD to reduce a second PSD. These reductions fall into three kinds, depending on whether the PSD is an equality, a single disjunct which is not an equality, or a PSD which has two or more disjuncts. The first kind depends on the fact that the terms which appear in SAM V are given a well-ordering. Equalities in SAM V are always written so that the left side of the equality is at least as high in this ordering as the right side of the equality. This ordering of the terms in an equality is convenient for both reduction and expansion. The well-ordering is described at the end of this section.

An a=b reduction of c is obtained by replacing all instances of a in c by the appropriate instance of b . Under the assumption that lowness in this ordering can be equated with simplicity, this type of reduction is in fact a simplification. In the second kind of reduction, where a PSD consisting of a single disjunction b reduces a PSD c , instances of b are applied to the disjuncts of c in order to replace them where possible by either TRU or FAL , a disjunct of c being replaced by TRU(FAL) if it is an instance (negation of an instance) of an instance of b . In the third kind of reduction, where a PSD b has n disjuncts (n 1), a PSD c is replaced by TRU if n disjuncts from c , considered as a PSD, form an instance of b .

In a similar classification expansions are of two types. The first type called self-expansion takes a single PSD and applies the following rules:

1. If the PSD has more than one disjunct, one of which has the form NOT(b=c), where b and c can be made to correspond by some instantiation of the PSD, a copy of the PSD is made with the equality deleted and the instantiation made in the copy.
2. If a PSD consists of a single disjunct of the form NOT(b=c) where b and c can be made to match by some instantiation, the main control of AUTO-LOGIC is notified that a contradiction has been obtained.
3. If a PSD has two disjuncts b and c which can be made identical by some instantiation, the instantiated copy of the PSD with one of the disjuncts deleted is generated.

The process of finding a common instance of two formulas we call matching. Matching is fundamental to several portions of AUTO-LOGIC and is described later.

The second type of expansion uses a PSD, b , to expand a second PSD, c . These expansions are exactly like the first two of the three reduction cases mentioned above except that an instantiation of c as well as b is required to make the corresponding reduction. In these cases an instantiated copy of c is made and the appropriate reduction applied. In addition, there is an operation of expanding with two multi-disjuncts. This operation, called resolution by some authors, generates a third multi-disjunct by joining appropriate instances of the original two multi-disjuncts and deleting two disjuncts of the form A and NOT(A) .

Digression is an attempt to use on a simple level the proof strategy of temporarily complicating a proof to gain some later simplification. Digression expands a formula d with an equality b=c by replacing an instance of the "simpler" term c in the formula d by the appropriate instance of b . When the result of this digression is brought up from the list of expansions, its progenitors and, in particular, b=c is not used to reduce the digression. If no other PSD's reduce the digression, it is deleted. If some reduction by a PSD other than b=c is possible, the digression is kept and business goes on as always. Hence, digression represents the use of "one step backward" in simplification.

We now describe the method currently implemented in AUTO-LOGIC, for applying these expansion processes. Two ordered lists of PSD's are kept during each phase of the algorithm. Initially, the first list, the list of reductions (LR), contains the original PSD's in the order given to AUTO-LOGIC. The second list, the list of expansions (LE), is initially empty. There is a main pointer called LOW which proceeds through LR starting at the top and proceeding down. At the top of the main loop, LOW distinguishes an element from the LR . The algorighm proceeds as follows: self-reduction is applied to the distinguished PSD, b . If b is reduced to FAL the algorithm halts and the fact that a contradiction has been reached is signalled. If b is reduced to TRU the LOW pointer is advanced and b is deleted. Otherwise, the formulas above b on LR are used, one at a time, to reduce b . If at any time b is reduced to TRU or FAL the appropriate action is taken. If b has not been reduced to TRU or FAL , then b is used to reduce the PSD's above b on LR . If one of these PSD's should be reduced to TRU it is then deleted; if reduced to FAL the algorithm signals the main control and is halted; but if reduced to some other PSD, this PSD is removed and inserted immediately below the distinguished PSD b . If no reduction is possible, expansions and digressions of b with the PSD's above b , and conversely, are generated. The new PSD's obtained by expanding and digressing are then placed on LE . The PSD's on LE are ordered by some criterion of potential utility. Various criteria have been tried with varying results. Finally, after all the reductions upward and expansions have been done, self-expansion is applied to the distinguished PSD and the results placed on LE .

LOW is then advanced. When LOW advances beyond the last PSD of LR , the PSD on the top of LE is removed and inserted at the bottom of LR and becomes the distinguished PSD. The algorithm then continues. If LE becomes exhausted the main control is so signalled and the algorithm halts.

For the purposes of defining the well-ordering of terms of SAM V, we consider the symbols of SAM V to be divided into two groups: constants and all other symbols. Within each group we consider the elements to be ordered by the magnitude of its numerical representation in SAM V. A term A is "greater than" B in the well-ordering of terms in case

a) A contains more occurrences of some constant c than B, and both terms have exactly the same number (possibly zero) of occurrences of each of the constants "greater than" c, or

b) A and B have exactly the same constants each with the same number of occurrences, but A follows B in the lexicographic order where all constants are considered to be greater than all non-constants and all non-constants are assumed to be in the same order position, or

c) The "constant structure" of A and B is the same but A is bigger than B in the lexicographic order (i.e., there is some non-constant c in A and d in B and A and B are identical up to occurrences at c and d, but the numerical representation of c is greater than that of d).

This well-ordering of terms has several interesting properties. First of all it guarantees that a re-lettering of the variables of a term will not drastically change its position in the well-ordering. Secondly, by using constants of different relative "size" we can give preference to one concept over another. As an application of this it is convenient to symbolize a newly defined concept with a symbol that is relatively high compared to the symbols used in the definition. While the basic theorems are being proved for the new concept, it is important that terms containing the new symbol be replaced by their definition. Once all the basic theorems concerning that concept are proved, we replace the new symbol by a symbol that is small relative to the symbols used in its definition. Then terms corresponding to the definitions can be "simplified" to a term involving the new symbol by using the basic theorems which have just been proven.

We close this section with a description of matching. Two formulas are said to match if they have a substitution instance in common. For example, the formulas Q(a, x) and Q(y, H(y, z)) match because the formula Q(a, H(a, z)) (called a matching formula) may be gotten by appropriate substitutions in either of them. In this particular example, Q(a, H(a, z)) is in fact a general matching formula, since all other matching formulas for the original pair may be obtained from it by substitution. The process of matching, i.e., of obtaining a general matching formula for two given expressions, is a basic tool in the construction of proofs. Consequently, much effort has been devoted to developing match algorithms and implementing them in SAM.

The fundamental match algorithm in AUTO-LOGIC is described below:

Step 1 Consider B and C as being stored at lines (1) and (2) respectively. Reletter the variables of line (2) so that it has no variables in common with line (1).

Step 2 Let us denote the n-th symbol -- ignoring parentheses and commas -- of line (1) by $(1)_n$. Similarly we define $(2)_n$.

Case a) If lines (1) and (2) are identical, the algorithm outputs (1) and stops.

Case b) Suppose n is the smallest integer such that $(1)_n$ is different from $(2)_n$. Since wffs are involved and Case a) does not hold, neither $(1)_n$ nor $(2)_n$ can be vacuous. We consider four subcases:

i) Suppose $(2)_n$ is a variable, say x, while $(1)_n$ is a function or individual constant. Then call D the unique subformula of (1) starting at $(1)_n$. If D contains x, output DOES NOT MATCH and stop. If D does not contain x, substitute D for x everywhere in (1) and (2). Go back and repeat Step 2.

ii) Proceed as in i) if the roles of (1) and (2) are interchanged.

iii) If $(1)_n$ and $(2)_n$ are different variables, replace $(2)_n$ everywhere in (1) and (2) by $(1)_n$.

iv) If $(1)_n$ and $(2)_n$ are different constants, output DOES NOT MATCH and stop.

Examples

Let us apply matching to $P(G(G(x, G(y, x)), z))$ and $P(G(G(x, y), G(x,x)))$.

(1) PGG x G y x z
(2) PGG uvG uu

(1) PGG x Gy x z
(2) PGG x v G x x

(1) PGG x G y x z

(2) PGG x G y x G x x

(1) PGG x Gy x G x x

(2) PGG x G y x G x x

Then $P(G(G(x, G(y, x)), G(x, x)))$ is the output of the algorithm and is in fact a general matching formula for the two wffs.

Let us apply matching to $Q(x, x)$ and $Q(y, H(y))$.

(1) Qxx

(2) Qy Hy

(1) Qxx

(2) Qx Hx

(1) DOES NOT MATCH

The variable x cannot be replaced by $H(x)$.

For a proof that this algorithm actually does produce a general matching formula or a "does not match" response in finitely many steps, see [2], pp. 26-27.

The above process, although quite helpful in some instances, cannot by itself come up with many of the matches one would like to be able to find. Suppose, e. g., that P is a commutative function. Clearly the two formulas $P(a, x)$ and $P(g(y), y)$ will match under this assumption -- $P(G(a), a) = P(a, G(a))$ is a general matching formula -- but the algorithm will fail. The obvious thing to do here is commute the terms in one of the expressions and then apply the algorithm, but the trick becomes a little more difficult when one tries to match longer, more complicated formulas under the assumption of commutativity. Matching with associativity gives rise to similar problems.

Recently, a general method for matching expressions involving commutative and/or associative functions has been worked out on paper and incorporated into SAM. It works by first taking the two formulas to be matched as they stand and applying the fundamental algorithm. If a general matching formula is not found in this first attempt, the arguments of all the associative functions are associated to the right and another try is made. The process continues in this way, associating and commuting arguments according to a prescribed pattern and invoking the algorithm at each step, until either a general matching formula is found or all the permissible rearrangements of terms are exhausted, in which case a definitive "does not match" response is given. The crux of the problem was, of course, the discovery of a method for generating permutations of terms in such a way that all allowable regroupings and reorderings would be gotten eventually with a minimum of repetitions.

To invoke this new matching capability, the operator merely inserts labels for the commutative and/or associative functions of his system in special program locations. A certain degree of control over matching and instantiation (a special case of matching in which the variables in one of the formulas to be matched are treated as constants) is possible through the setting of limits on the amount of machine time SAM is to spend applying either process to a pair of formulas. Usually, the timers for matching and instantiation are set for two seconds and one second, respectively, but they may be varied at will depending on the characteristics of the mathematical system being investigated. If long, difficult matches and instantiations are expected to be important in the proof of theorems, the timers can be set higher. Setting them lower, on the other hand, forces SAM to de-emphasize these two processes in its development of results. In short, the implementation of commutative-associative matching offers the experimenter yet another means of interacting with SAM, as well as a helpful new method for generating theorems.

Despite the successes that have been achieved to date, it would be wrong to say that the matching procedure which SAM currently employs is an optimal one. Formula pairs involving commutative and associative functions frequently have several general matching formulas, but at present SAM finds only one of these. Right now, this is not a major handicap, but it will certainly have to be overcome before SAM is able to consider axiom systems in which free function variables are present. Fortunately, the theoretical groundwork for a more complete extension current matching techniques has already been laid, and implementation of same is now in progress. A thorough treatment of matching in omega-order logic, including many of the problems we are dealing with at the moment, may be found in [5] .

SECTION III

CONTROL, INPUT/OUTPUT

In this section we describe how SAM operates as a running program and, in particular, the man-machine interface. In conceptual and programming terms this breaks up into the two aspects of control and input/output. At the beginning of this year it seemed reasonable to expect that we would soon wish to have sophisticated language facilities for man-machine communication and activity at the level of full proofs in a natural deduction calculus. However, experience quickly showed that AUTO-LOGIC was so powerful and flexible that it would be very worthwhile to work extensively with man-machine interaction at the AUTO-LOGIC level. Hence, operation of SAM has evolved into a mode in which the user is monitoring the lists of reductions and expansions as the AUTO-LOGIC algorithm is transforming them. The user may intervene by stopping the algorithm process and himself adding or modifying the lists of formulas. The use of the CRT display and PDP-6 time sharing system are of course essential to these activities. It seems unlikely that SAM V could have grown by a process of evolution as it has without the use of the time shared system to make programming, debugging, and in-core modification feasible and rapid.

In the current implementation of SAM the user initiates action by setting up a list of PSD's as the initial list of reductions (with a void LE). Some of the formulas may be marked with an asterisk to indicate that they are the negations of formulas whose proof is sought. AUTO-LOGIC is then turned loose to generate consequences of these initial formulas. If none of the original formulas are starred then the results are consequences of the original formulas, presumably axioms or previously proved theorems. If some of the original formulas are starred then all consequences of the original starred formulas are starred and it is hoped that SAM will derive a contradiction (prove FAL). In that event SAM has demonstrated by contradiction that the disjunction of the unnegated versions of the original starred formulas is a logical consequence of the other initial formulas (the latter are usually axioms and previously proved theorems).

As AUTO-LOGIC works on the lists of expansions and reductions, the user is able to watch these lists on the CRT display. He sees formulas appearing on the list of reductions, being reduced to simpler form, reducing other formulas, generating formulas for the list of expansions, and disappearing in favor of more powerful formulas. In this monitoring process the user gains great insight into the logical processes of SAM. It is like having a window on the thoughts of a powerful but very different kind of mathematician. Such an understanding of AUTO-LOGIC as a dynamic entity is very important for finding useful improvements to make in the basic algorithm. It is also important in allowing the user to interact with a given proof.

The user may intervene in the process of proof in a number of ways. His selection and ordering of the initial formulas is of course an important factor in determining the course that AUTO-LOGIC will take. Too many or ill chosen sets of initial formulas tend to send AUTO-LOGIC off proving trivial and uninteresting results without ever getting to interesting formulas. From a good starting point AUTO-LOGIC will produce useful and interesting results. As the user sees that

AUTO-LOGIC is running out of useful things to do with the formulas first given, he can halt the process and insert additional axioms cr other material. He can also guide tne process by deleting formulas which seem unimportant or distracting. This real time interplay between man and machine has been found to be an exciting and rewarding mode of operation.

Since formulas both appear and disappear in the process of proof, a record is kept of each formula as it comes under consideration. This record yields a history of a session which can be used for later analysis and review. A program called HIST can step through this history and display on the scope only those steps actually used in the proof of a sequence pointed at by the light pen.

In operating SAM, the user sits at a display-teletype console. The display shows a section of the proof and a set of buttons. By using the light pen on the display buttons the user controls the display and the action of SAM. He can cause any section of the proof to be displayed, have the proof "roll" by, or track on the end of the reductions -- i.e., display the lowest formulas on the list. The display is updated every second to show the current proof status. The user can halt the proof procedure, continue, indicate a desire to insert or delete formulas, request a proof history, enter the debugging mode, and request a print out. For insertion or deletion of formulas the user indicates position by light pen and, for insertions, types the desired formula.

In Section I we described the symbols and formulas of SAM as they appear in their simplest form -- symbols as single letters with subscripts and formulas as strings of symbols with parentheses showing complete structure. Simple routines have been written for input and output of formulas in this notation. Such routines are convenient for debugging and preliminary experimentation. But for more effective man-machine communication we require input/output in notation closer to that in common use by mathematicians. There are two aspects to this requirement, the need for a large set of symbols with such features as varying size and position, e.g., subscripts and superscripts, and the need for flexibility in the format of formulas.

At the symbol level we are concerned with two distinct situations. Hard copy can be quickly and easily produced on teletype and line printer. With the line printer even very large sets of formulas can be printed rapidly. However, the set of characters and the output format available are limited and modifications are expensive and slow. On the other hand, the CRT display and incremental plotter can accommodate any symbols and formats that the user cares to specify. However, the CRT display cannot easily produce hard copy of high quality and the plotter is not a fast economical device for producing hard copy. With these considerations in mind, input/output for SAM has been programmed to deal with symbols in three modes.

Internally a symbol is simply a number. The first output mode, the character-subscript mode, treats this number as a direct coding of the symbol as a sequence of ASCII characters. It is useful representation when working on the programming of SAM since output may be quickly produced on the line printer and the correspondence between internal and external form is constant. The second mode, the actual drawing of symbols, associates with each internal code a special symbol which is drawn in the form that the user wishes. For convenience, flexibility and efficiency, the correspondence between number and form is kept entirely arbitrary - a simple table in SAM contains all the information on the correspondence. This table does not contain actual instructions for drawing symbols but transliterated codes for the symbol. For example, the Greek letters α and β appear as ↑GA and ↑GB , the mathematical symbols $\cup$ and $\cap$ appear as \J and \M . With this transliteration approach the actual service routines for I/O devices can be written completely independently. The use of transliteration codes gives us a third mode of output, that in which symbols are represented by their transliterations. This mode can actually be used to produce quick output on the line printer for debugging or cursory examination. More important, it allows for the input of arbitrary formulas from the teletype. In addition, formulas in transliteration can be stored in machine readable form.

Since the formulas present in AUTO-LOGIC contain few logical connectives and are usually short, the conveniences of notation which we have implemented are restricted to the presentation of central binary connectives (such as + and =) between their arguments and the suppression of some unnecessary parentheses.

In actual operation, the I/O modes are used as follows. Formulas for display on the scope are transliterated and passed through a display service routine which produces suitable display instructions.

Formulas for hard copy output are written out on tape, drum, line printer or teletype in the form that the user wishes, transliterated or character-subscript. From tape or drum, output may be later produced on line printer or plotter. Plotter output is produced from transliterated formulas in a tape or drum file by a small independent program. Thus it does not slow down the use of SAM nor waste space in the program.

SECTION IV

EXPERIMENTATION WITH SAM V

In this section we describe some of the experimentation that has been done with SAM V. Along with development of the basic program, experimentation with SAM V has been a continuing activity. Such experimentation serves to determine whether or not SAM V is progressing toward the ultimate goal of being a program which can be of practical utility in proving theorems of real mathematical substance. In addition experimentation with SAM indicates the features which are in need of improvement and thus serves as a basis for plans for the future expansion.

In the first part of this section we discuss experimentation with abstract algebra, group theory in particular. Experimentation along these lines has demonstrated that the techniques of AUTO-LOGIC in handling equality and algebraic notions are very powerful. In the second part, we report on an exciting result of current experimentation, the actual solution by man-machine interaction of an open problem in the field of lattice theory. This result, called SAM's Lemma, can be viewed as a rudimentary demonstration of the great utility of the man-machine approach to the automation of mathematics and as an actual sample of a program which partially realizes such aspirations. In the third part we present some examples of experimentation done with the simple sort structure that has been added to the repertoire of SAM.

The AUTO-LOGIC algorithm described in Section II seems to be quite successful in cases where the PSD's are equalities. We give an example from group theory below in two different forms. We take quantifier-free axioms for group theory which say that for the group multiplication, $\alpha \circ \beta$ that $-1(\alpha)$ is the left inverse and A is the left identity. In the first run we derive some consequences of these axioms (See Figure 1). In addition, we have printed out the history of the proofs of four of the more interesting consequences (See Figures 2 through 5. In Figure 6 we list those PSD's which SAM generated but was unable to reduce immediately to TRU). This should help explain the subject matter of Section II. In the second run we insert the negation of the statement that V is also a right inverse and that A is also a right identity and that $-1(\alpha)$ $-1(-1(\alpha)) = \alpha$ (See Figure 7) This shows the second mode in which SAM can operate.

00001 $(\alpha \circ \beta) \circ \gamma = \alpha \circ (\beta \circ \gamma)$ AXM

00002 $^{-1}(\alpha) \circ \alpha = A$ AXM

00003 $A \circ \alpha = \alpha$ AXM

00004 $(^{-1}(\alpha) \circ \alpha) \circ \alpha_1 = \alpha_1$ 00002 D! OF 00003 (01)

00005 $^{-1}(\alpha) \circ (\alpha \circ \alpha_1) = \alpha_1$ 00001 RED OF 00004 (02)

00011 $^{-1}(^{-1}(\alpha_1)) \circ A = \alpha_1$ 00002 EXP OF 00005 (03)

00012 $^{-1}(^{-1}(^{-1}(\alpha_1))) \circ \alpha_1 = A$ 00005 EXP OF 00011 (04)

00013 $^{-1}(^{-1}(^{-1}(^{-1}(\alpha_1)))) \circ A = \alpha_1$ 00005 EXP OF 00012 (05)

00014 $^{-1}(^{-1}(\alpha_1)) = \alpha_1$ 00011 RED OF 00013 (06)

00016 $\alpha_1 \circ {}^{-1}(\alpha_1) = A$ 00002 EXP OF 00014 (07)

00017 $(\alpha_1 \circ {}^{-1}(\alpha_1)) \circ \alpha = \alpha$ 00003 DI OF 00016 (10)

00023 $\alpha_1 \circ (^{-1}(\alpha_1) \circ \alpha) = \alpha$ 00001 RED OF 00017 (11)

Figure #1

```
00001   (α ∘ β) ∘ γ = α ∘ (β ∘ γ)                        AXM
00002   -1(α) ∘ α = A                                     AXM
00003   A ∘ α = α                                         AXM
00004   (-1(α) ∘ α) ∘ α1 = α1              00002 DI OF 00003 (01)
00005   -1(α) ∘ (α ∘ α1) = α1              00001 RED OF 00004 (02)
00011   -1(-1(α1)) ∘ A = α1                00002 EXP OF 00005 (03)
00012   -1(-1(-1(α1))) ∘ α1 = A            00005 EXP OF 00011 (04)
00013   -1(-1(-1(-1(α1)))) ∘ A = α1        00005 EXP OF 00012 (05)
00014   -1(-1(α1)) = α1                    00011 RED OF 00013 (06)
00015   α1 ∘ A = α1                        00014 RED OF 00011 (07)
```

Figure #2

00001 $(\alpha \circ \beta) \circ \gamma = \alpha \circ (\beta \circ \gamma)$ AXM

00002 $^{-1}(\alpha) \circ \alpha = A$ AXM

00003 $A \circ \alpha = \alpha$ AXM

00004 $(^{-1}(\alpha) \circ \alpha) \circ \alpha_1 = \alpha_1$ 00002 DI OF 00003 (01)

00005 $^{-1}(\alpha) \circ (\alpha \circ \alpha_1) = \alpha_1$ 00001 RED OF 00004 (02)

00011 $^{-1}(^{-1}(\alpha_1)) \circ A = \alpha_1$ 00002 EXP OF 00005 (03)

00012 $^{-1}(^{-1}(^{-1}(\alpha_1))) \circ \alpha_1 = A$ 00005 EXP OF 00011 (04)

00013 $^{-1}(^{-1}(^{-1}(^{-1}(\alpha_1)))) \circ A = \alpha_1$ 00005 EXP OF 00012 (05)

00014 $^{-1}(^{-1}(\alpha_1)) = \alpha_1$ 00011 RED OF 00013 (06)

00016 $\alpha_1 \circ {}^{-1}(\alpha_1) = A$ 00002 EXP OF 00014 (07)

Figure #3

00001 $(\alpha \circ \beta) \circ \gamma = \alpha \circ (\beta \circ \gamma)$ AXM

00002 $^{-1}(\alpha) \circ \alpha = A$ AXM

00003 $A \circ \alpha = \alpha$ AXM

00004 $(^{-1}(\alpha) \circ \alpha) \circ \alpha_1 = \alpha_1$ 00002 DI OF 00003 (01)

00005 $^{-1}(\alpha) \circ (\alpha \circ \alpha_1) = \alpha_1$ 00001 RED OF 00004 (02)

00011 $^{-1}(^{-1}(\alpha_1)) \circ A = \alpha_1$ 00002 EXP OF 00005 (03)

00012 $^{-1}(^{-1}(^{-1}(\alpha_1))) \circ \alpha_1 = A$ 00005 EXP OF 00011 (04)

00013 $^{-1}(^{-1}(^{-1}(^{-1}(\alpha_1)))) \circ A = \alpha_1$ 00005 EXP OF 00012 (05)

00014 $^{-1}(^{-1}(\alpha_1)) = \alpha_1$ 00011 RED OF 00013 (06)

Figure #4

```
00001   (α ∘ β) ∘ γ = α ∘ (β ∘ γ)                          AXM

00002   ~(α) ∘ α = A                                       AXM

00003   A ∘ α = α                                          AXM

00004   (~(α) ∘ α) ∘ α1 = α1                 00002 DI OF 00003 (01)

00005   ~(α) ∘ (α ∘ α1) = α1                 00001 RED OF 00004 (02)

00006   ~(A) ∘ α1 = α1                       00003 EXP OF 00005 (03)

00007   ~(~(A)) ∘ α1 = α1                    00005 EXP OF 00006 (04)

00010   ~(A) = A                             00002 EXP OF 00007 (05)
```

Figure #5

00001 $(\alpha \circ \beta) \circ \gamma = \alpha \circ (\beta \circ \gamma)$ AXM

00002 ${}^{-1}(\alpha) \circ \alpha = A$ AXM

00003 $A \circ \alpha = \alpha$ AXM

00004 $({}^{-1}(\alpha) \circ \alpha) \circ \alpha_1 = \alpha_1$ 00002 DI OF 00003 (01)

00005 ${}^{-1}(\alpha) \circ (\alpha \circ \alpha_1) = \alpha_1$ 00001 RED OF 00004 (02)

00006 ${}^{-1}(A) \circ \alpha_1 = \alpha_1$ 00003 EXP OF 00005 (03)

00007 ${}^{-1}({}^{-1}(A)) \circ \alpha_1 = \alpha_1$ 00005 EXP OF 00006 (04)

00010 ${}^{-1}(A) = A$ 00002 EXP OF 00007 (05)

00011 ${}^{-1}({}^{-1}(\alpha_1)) \circ A = \alpha_1$ 00002 EXP OF 00005 (03)

00012 ${}^{-1}({}^{-1}({}^{-1}(\alpha_1))) \circ \alpha_1 = A$ 00005 EXP OF 00011 (04)

00013 ${}^{-1}({}^{-1}({}^{-1}({}^{-1}(\alpha_1)))) \circ A = \alpha_1$ 00005 EXP OF 00012 (05)

00014 ${}^{-1}({}^{-1}(\alpha_1)) = \alpha_1$ 00011 RED OF 00013 (06)

00015 $\alpha_1 \circ A = \alpha_1$ 00014 RED OF 00011 (07)

00016 $\alpha_1 \circ {}^{-1}(\alpha_1) = A$ 00002 EXP OF 00014 (07)

00017 $(\alpha_1 \circ {}^{-1}(\alpha_1)) \circ \alpha = \alpha$ 00003 DI OF 00016 (10)

00020 $\alpha_1 \circ ({}^{-1}(\alpha_1) \circ \alpha) = \alpha$ 00001 RED OF 00017 (11)

00021 $\alpha \circ (\beta \circ {}^{-1}(\alpha \circ \beta)) = A$ 00001 EXP OF 00016 (10)

Figure #6

00001 •	NOT $(C_2 \bullet {}^{-1}(C_2) = A)$ NOT $(C_1 \bullet A = C_1)$ NOT $({}^{-1}({}^{-1}(C_3)) = C_3)$	AXM
00002	$A \circ \alpha = \alpha$	AXM
00003	${}^{-1}(\alpha) \circ \alpha = A$	AXM
00004	$(\alpha \circ \beta) \circ \gamma = \alpha \circ (\beta \circ \gamma)$	AXM
00005	$({}^{-1}(\alpha_1) \circ \alpha_1) \circ \alpha = \alpha$	00002 DI OF 00003 (01)
00006	${}^{-1}(\alpha_1) \circ (\alpha_1 \circ \alpha) = \alpha$	00004 RED OF 00005 (02)
00012	${}^{-1}({}^{-1}(\alpha)) \circ A = \alpha$	00003 EXP OF 00006 (03)
00013	${}^{-1}({}^{-1}({}^{-1}(\alpha))) \circ \alpha = A$	00006 EXP OF 00012 (04)
00014	${}^{-1}({}^{-1}({}^{-1}({}^{-1}(\alpha)))) \circ A = \alpha$	00006 EXP OF 00013 (05)
00015	${}^{-1}({}^{-1}(\alpha)) = \alpha$	00012 RED OF 00014 (06)
00016	$\alpha \circ A = \alpha$	00015 RED OF 00012 (07)
00017 •	NOT $(C_2 \circ {}^{-1}(C_2) = A)$ NOT $(C_1 \circ A = C_1)$ NOT $(C_3 = C_3)$	00015 RED OF 00001 (07)
00020 •	NOT $(C_2 \circ {}^{-1}(C_2) = A)$ NOT $(C_1 \circ A = C_1)$	RED OF 00017 (10)
00021 •	NOT $(C_2 \circ {}^{-1}(C_2) = A)$ NOT $(C_1 = C_1)$	00016 RED OF 00020 (11)
00022 •	NOT $(C_2 \circ {}^{-1}(C_2) = A)$	RED OF 00021 (12)
00023	$\alpha \circ {}^{-1}(\alpha) = A$	00003 EXP OF 00015 (07)
00024 •	NOT $(A = A)$	00023 RED OF 00022 (13)
00025 •	FAL	RED OF 00024 (14)

Figure #7

Experimentation with SAM V and AUTO-LOGIC produced an important result, one which we found both exciting and encouraging - the proof of a previously unresolved open problem. A mathematician - one with an intimate knowledge of the innards of SAM - and SAM V obtained the proof in a significant display of man-machine cooperation. Preliminary work was being done in the theory of modular lattices with a partial goal being to see whether SAM V could be guided to a proof of the results in [6]. In addition, it seemed possible to hope for a later attack on the unresolved problem presented there. Long before it seemed likely that enough development had been made along elementary lines it was noted that AUTO-LOGIC had proven a formula from which a positive solution was an immediate consequence. That formula was a crude form of what we now call SAM's Lemma. In evaluating the significance of this demonstration it is important to note the interactive aspects of its construction. The mathematician was guiding SAM in the broad lines of development of the theory of modular lattices and was present to notice a useful intermediate result. At the same time, the algorithms of AUTO-LOGIC were working to generate results that might be useful without getting lost in a mass of trivial and nearly equivalent formulas. SAM V was not capable of understanding all the consequences of many of the proven formulas but the mathematician despite a reasonable amount of prior effort, had not been able to see the key steps required to obtain a useful lemma, 'SAM's Lemma.

At this stage in the development of SAM we are always careful to check the results of automatic proofs. In this case the check verified the result and, as usually happens with first proofs in mathematics, led to a much more compact way of presenting the key ideas in the demonstration. The re-phrasing into common mathematical terms is given below along with a version of the proof by AUTO-LOGIC. The version of the semi-automatic proof that we give is somewhat shorter and more straightforward than the original since, with the benefit of hindsight, we were able to guide the process more directly to the Lemma. Note that the phrasing of some axioms is imposed by the search for simplicity in axioms (e.g., the associative law is SAM's choice of the simplest form in a context where commutativity is also present). Note also the following table of correspondence between the symbols of SAM and the notions mentioned below:

P	0 (first element)
Q	1 (last element)
D	meet (analogous to intersection and minimum)
A	join (analogous to union and maximum)
A1, B1	the a, b of [6]
R1, R2	$(a \vee b)'$, $(a \wedge b)'$
A2, B2	the x, y of [6]

Theorem 1, (Bumcrot [6]) If $(L, \leqq)$ is a modular lattice with 0 and 1 and if a, b in L are such that $a \vee b$ and $a \wedge b$ have (not necessarily unique) complements, then a and b have complements.

Theorem 2, (Bumcrot [6]) If $(L, \leqq)$ is a modular lattice with 0 and 1, if a, b in L have unique complements a', b' respectively, and if $a \vee b$ and $a \wedge b$ have complements, then $a' \vee b'$ is a complement of $a \wedge b$ and $a' \wedge b'$ is a complement of $a \vee b$.

Open Problem Bumcrot [6]) Under the hypotheses of Theorem 2 is it necessarily true that the complements of $a \vee b$ and $a \wedge b$ are unique?

<u>SAM's Lemma</u> Under the hypotheses of Theorem 1

$$(a \vee b)' = x \wedge y$$

dually

$$(a \wedge b)' = \bar{x} \vee \bar{y}$$

<u>Theorem (Oglesby, SAM V)</u> Under the hypotheses of Theorem 2, the complements of $a \vee b$ and $a \wedge b$ are unique.
The theorem follows immediately from SAM's Lemma since, by the assumed uniqueness of complements for a and b, $x \wedge y$ and $\bar{x} \vee \bar{y}$ are independent of which $(a \vee b)'$ and $(a \wedge b)'$ are used in their construction.

In Figures 8 and 9 we show an early proof of SAM's Lemma. The numbers at the left margin indicate the order in which SAM has added the formulas to the list of reductions, missing numbers correspond to formulas that SAM has eliminated in favor of combinations of simpler formulas. The numbers at the right margin indicate the depth of proof required. Note how the introduction of the associative axioms is deferred so that SAM can first work on the consequences of the other axioms.

In Figure 10 we show the history of a proof of SAM's Lemma on the latest version of SAM. In this proof $\vee$, $\wedge$, $\circ$ replace A, D, and P respectively. In addition, the functions $\vee$. $\wedge$ are given to SAM as being both associative and commutative so that the resulting proof is much simpler. In fact, the new proof looks deceptively simple.

PROOF OF SAM's LEMMA

```
0001   A(P,X) = X                                P=O                                      (00)
0002   D(P,X) = P                                Q=1                                      (00)
0003   A(Q,X) = Q                                                                         (00)
0004   D(X,A(X,Y)) = X                                                                    (00)
0005   A(X,D(X,Y)) = X                                                                    (00)
0006   D(Y,X) = D(X,Y)                           with 0611 & 0622 give lattice            (00)
0007   A(Y,X) = A(X,Y)                                                                    (00)
0010   NOT (D(X,Z) = X)                                                                   (00)
       D(Z,A(X,Y)) = A(X,D(Y,Z))
0011   NOT (D(X,Z) = Z)                                   modular lattice                 (00)
       D(X,A(Y,Z)) = A(Z,D(X,Y))
0012   D(Q,X) = X                                                                         (00)
0013   D(X,X) = X                                0004 EXP OF 0005                         (01)
0014   A(Y,Y) = Y                                0005 EXP OF 0013                         (02)
0066   D(B1,B2) = P                                                                       (00)
0067   A(B1,B2) = Q                              Implied by 0072-0100                     (00)
0070   D(A1,A2) = P                              (SAM was able to prove these             (00)
0071   A(A1,A2) = Q                               and did so earlier)                     (00)
0072   A(R2,D(A1,B1)) = Q                        R2 a complement of                       (00)
0073   D(A1,D(B1,R2)) = P                           D(A1, B1)                             (00)
0074   A(A1,A(B1,R1)) = Q                        R1 a complement of                       (00)
0075   D(R1,A(A1,B1)) = P                           A(A1, B1)                             (00)
0076   A(R1,D(A1,R2)) = B2                       Def. of A2 & B2                          (00)
0100   A(R1,D(B1,R2)) = A2                                                                (00)
0542   D(R1,A2) = R1                             0004 EXP OF 0100                         (01)
0543   D(R1,B2) = R1                             0004 EXP OF 0076                         (01)
0551   NOT (D(Y,Z) = Z)                          0007 RED OF 0550                         (03)
       A(Y,Z) = Y
```

Figure #8

```
0555     NOT (A2 = P)                 0070 RED OF 0554          (05)
         A1 = Q

0557     NOT (B2 = P)                 0066 RED OF 0556          (05)
         B1 = Q

0563     NOT (D(A1,A(B1,R1)) = A(B1,R1)) 0551 EXP OF 0074(04)
         A1 = Q

0564     NOT (D(B1,R2) = P)           0551 EXP OF 0073          (04)
         A(A1,D(B1,R2)) = A1

0600 *   D(B2,A(R1,Y)) = A(R1,D(B2,Y)) 0006 RED OF 0577         (04)

0611     A(Z,A(X,Y))=A(X,A(Y,Z))                                (00)

0617     A(R1,B2) = B2                0014 RED OF 0616          (10)
0621     NOT (A(A1,B1) = P)           0074 RED OF 0620          (06)
         R1 = Q

0622     D(Z,D(X,Y))=(D(X,D(Y,Z))                               (00)

0630     NOT (D(A1,B1) = P)           0073 RED OF 0627          (06)
         R2 = Q

0634     NOT (D(B1,D(R1,R2))=D(B1,R2))0006 RED OF 0633          (06)
         A2 = R1

0636     D(R1,D(B2,Z))=D(R1,Z)        0006 RED OF 0635          (03)

0637     NOT (R2 = R1)                0636 RED OF 0575          (06)
         A(B2,D(R1,Y))=R1

0641     D(R1,A(B2,Y))=R1             0543 RED OF 0640          (05)

0643     D(R1,D(A2,Z))=D(R1,Z)        0006 RED OF 0642          (03)

0645     D(R1,A(A2,Y)) = R1           0542 RED OF 0644          (05)

0650     D(A1,D(A2,Z))=P              0002 RED OF 0647          (03)

0653     D(B1,D(B2,Z))=P              0002 RED OF 0652          (03)

0656**   D(A2,B2) = R1                0001 RED OF 0655
```

* Key result
** SAM's Lemma

Figure #9

00001 $(Y \vee X) \wedge X = X$ AXM

00002 $(Y \wedge X) \vee X = X$ AXM

00003 $\sim (Z \wedge X = X)$
$(Z \wedge Y) \vee X = (Y \vee X) \wedge Z$ AXM

00005 $\circ \wedge X = \circ$ AXM

00012 $(A_2 \wedge B_1) \vee A_1 = A_2$ AXM

00013 $(A_2 \wedge A_1) \vee A_1 = B_2$ AXM

00015 $A_2 \wedge A_1 = \circ$ AXM

00027 $A_2 \wedge A_1 = A_1$ 00001 EXP OF 00012 (01)

00032 $\circ \vee X = X$ 00002 EXP OF 00005 (01)

00036 $(A_2 \wedge Y) \vee A_1 = (A_1 \vee Y) \wedge A_2$ 00003 EXP OF 00027 (02)

00045 $(A_2 \wedge (A_2 \wedge A_1)) \vee A_1 = B_2 \wedge A_2$ 00013 EXP OF 00036 (03)

00046 $(A_2 \wedge \circ) \vee A_1 = B_2 \wedge A_2$ 00015 RED OF 00045 (04)

00047 $A_1 \vee \circ = B_2 \wedge A_2$ 00005 RED OF 00046 (05)

00050 $B_2 \wedge A_2 = A_1$ 00032 RED OF 00047 (06)

Figure #10

SECTION V

TROLL

A. Introduction

The coding for SAM V has been done in machine language to take advantage of the large flexible instruction set available on the PDP-6 and to achieve the speed and efficiency necessary to the practical operation of SAM. The basic framework of the programming of SAM has been a list processing language which we have named TROLL (Threaded, Ringed, Oriented List Language). This list processing language is of interest as a separate entity in its own right. It is a general purpose list language of the type which is most useful for the purposes of SAM. Within the requirements of storage and linking information required by SAM it is a most efficient variety of link processing language.

TROLL is a set of list processing subroutines designed to be embedded in FORTRAN or a machine language. It could consist of a set of primitives, coded in machine language for a particular machine, and a set of routines canonically written in FORTRAN, but in actuality coded in machine language for added efficiency. The primitives could be used to fetch values from fields within list cells, and to store values within these fields. Obviously by changing the primitives one can greatly change the nature of the list processor. In our implementation all the routines are written in MACRO-6, the PDP-6 assembly language.

TROLL is threaded in that the last cell of a simple sublist links back to its reference. Thus a pushdown stack is not needed when searching structurally through a simple list. TROLL is ringed (or knotted) in that it is possible to have multi-referenced sublists. Of course in searching through a list with multi-referenced sublists, a pushdown memory is needed in order to come back up through the structure. This is provided in one set of search functions. TROLL, unlike a symmetric list language, is oriented in that there is a preferred left to right, top to bottom direction for lists.

B. Definitions

A list is either simple or multi-referenced.

A simple list is a list that is referenced exactly once.

A multi-referenced list is a list that may be referenced zero, one, or many times. A reference count, which is the number of times that the multi-reference list is referred to, is kept in a designated item in the list called the header. For most purposes the header looks like a one item list which refers to the simple list which is the body of the multi-referenced list. Each item of a list is stored in a cell, the exact nature of which is determined by the particular implementation of TROLL.

A datum cell is a cell containing a quantity of information, the size depending on the particular implementation of TROLL.

An address cell is a cell containing a link to an arbitrary data structure.

A reference cell is a cell containing a link to a simple list.

A *multi-reference cell* is a cell containing a link to a multi-referenced list.

A *sublist* is a list which is linked to by a reference cell or a multi-reference cell in another list.

C. Cells

Cells contain the following information:

The *terminal* or *T* field, which contains 1 if the cell is the last cell of a sublist (0 otherwise).

The *code* or *C* field, which contains:

0 if the cell contains a datum,
1 if an address,
2 if a reference, and
3 if a multi-reference.

The *datum* or *D* field, which contains the datum of the cell.

The *link* or *L* field, which contains the address, reference, or multi-reference

The *head* or *H* field, which contains 1 if the cell is a header, 0 otherwise.

The *count* or *N* field, which contains the reference count.

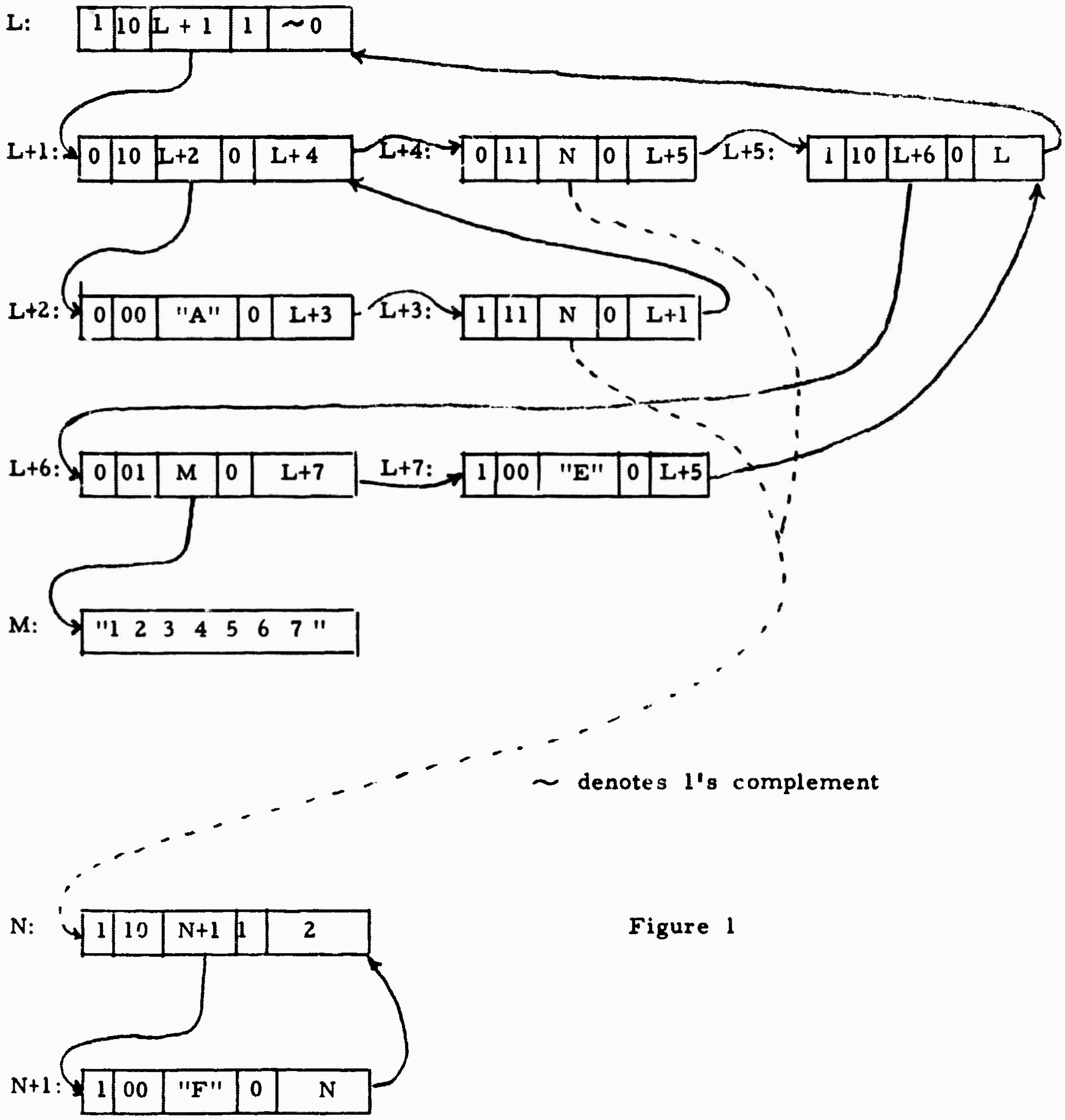
The *successor* or *S* field which contains a link to the cell to the right of the current cell.

In a header cell, the datum and successor fields are undefined. In a non-header cell the count field and either the datum or link field is undefined.

On the PDP-6 a cell is a single 36-bit machine word with the fields stored as follows:

T	C	D or L	H	N or S
0 1	2	3 17	18	19 35

When the N field exists (i.e., bit 18=1), the one's complement of the reference count is stored there. Otherwise the fields contain their actual values as described above.



~ denotes 1's complement

Figure 1

D. More Definitions

There are five basic types of cells:

Datum or D cells

Reference or R cells

Empty or E cells - (special case of reference cells, with pointer field of zero)

Address or A cells

Multi-reference or M cells

There are four groupings of cells:

Word or W cells (any cell)

Element or E cells (datum or address)

Name or N cells (reference, multi-reference or header)

Extreme or X cells (furthest cell in a given direction on a list of sublist)

From a given cell there are four possible directions:

Up - U

Down - D

Left - L

Right - R

Relative to a cell, there are four possible locations:

Top (T) of sublist

Bottom (B) of sublist

Left (L)

Right (R)

E. Basic TROLL Programs

Look functions: L * ** (IP)

where * is a direction and ** is a grouping. They search a list starting at IP, in the direction indicated, until a cell of the required grouping is found. They normally return an unflagged pointer to the quantity found; return a pointer to the header and a flag of 3 if a header is found; return a pointer with a flag of 2 to the reference cell if the argument of LR ** is the last cell of a sublist or if the argument of LL ** is the first cell of a sublist; return a flagged pointer to the multi-reference cell if LU ** or LD ** requires going into the multi-referenced sublist.

In the PDP-6 version of LDW , LDE , LDN , if an address is found in bits 3-17 of the pointer, the cell with this address functions, for the look, as a header; that is, if a look reaches this cell by way of the successor field of the previous cell, a pointer with a flag of 2 is returned. Bits 3-17 is called a stop address.

New functions: N * ** (IP, I)

where * is a basic type, ** is a relative location. The quantity I is inserted into a new cell in the relative location, and a pointer to this new cell is returned.

ND ** (IP, I) the datum I is placed in the new cell.

NR ** (IP, I) I is a pointer to an unreferenced header. The header is erased and a reference to the body of the list is created.

NE ** (IP) an empty reference is created. These functions have only the first argument since no quantity is inserted into the cell.

NA ** (IP, I) the quantity I is placed into a new cell, and a new address cell is created, pointing to the copy of I .

NM ** (IP,I) The reference count on the header pointed to by I is increased by I .

N * T(IP,I) and
N * B(IP,I) require IP to point to a name cell.

Instead functions: IN * (IP,I)
where * is a basic type. Instead functions erase the contents of the cell to which IP points and inserts the quantity I in a similar manner to the new functions. They return the old datum, or quantity pointed at by an address, or -1 if the original contents were a header or reference, or -2 if the original contents were a multi-reference.

Multi-reference look functions: M * ** (IP)
where * is a direction, and ** is a grouping. These are similar to look functions. If a list contains no multi-referenced sublists, they are identical to look, returning an unflagged pointer to the cell found, and zero if the header of the main list is found. Multi-reference looks have a pushdown memory so that a look into a multi-referenced sublist, and return, is possible. If such a look is required, the proper addition to the pushdown list is made and the look continues. If an MR ** or ML ** requires coming out of a (multi-referenced) sublist, a flagged pointer to the multi-reference cell is returned.

F. List of Available Space

The list of available space (LAV) is a linear pushdown list of available cells. LAV is initialized by CALL LAS(IPA,IPB) where IPA , IPB are (inclusive) pointers to the ends of the block of memory to be initialized. The first cell of the available space block is initialized to a header for the list of available space. A pointer to this header is found in (global) location LAV. The second cell of the available space block is initialized to an empty header for the recursion pushdown list. A pointer to this header is found in (global) location LAVS. The rest of the available space block is initialized as the body of LAV, a linear list each element of which is a zero datum.

NUC(IX) returns a pointer to a new cell, the previous top cell of LAVS. IAC is updated. IX is a dummy argument.

ERACEL(IP) zeros cell IP, pushes it on LAVS, and updates IAC.

JNK(IP) makes a linear list out of the cell IP and any cell linked to by IP, links this linear list to the top of LAVS, updates IAC, and relinks the list in which IP originally occurred. If IP points to a header, the reference count is decreased by 1. If it now is negative, the whole list is erased. When a multi-reference cell is erased, the reference count is decreased by 1 and, if now negative, the whole multi-referenced sublist is erased.

NOTE: By changing the list pointed to by LAV one can use more than one list of available space.

H. Minor Routines

LVL(IP) returns pointer to reference cell for current sublist or flagged pointer if same is a header.

MOV(IP) returns IP. Removes IP from its list and makes it into a header. If IP not a reference cell, does nothing, returns zero.

LCOPY(IP) returns pointer to header of copy of IP . IP must be a header or reference.

MTH(IX) returns a pointer to an empty (pointer field = 0) header. IX is a dummy argument.

I. Examples

Let "adr" be a FORTRAN variable containing a pointer to adr . Referring to Figure 1:

Look LRX("L+1") returns L+5

LLW("L+7") returns L+6

LUE("L+4") returns L+2

LDN("L+3") returns L+4

LDE("L+3") returns L+4 flagged

LRW("L+5") returns 0

New NAL("L+3", 3275) inserts an address cell pointing to the word containing 3275, between L+2 and L+3.

NNR(L+2", N) inserts a multi-reference cell between L+2 and L+3 and increases the reference count of N to 3 (stored as 3).

NDT("L+5", P) inserts a datum cell pointing to P before L+6.

MTH The list (1, (2,3)) can be created by

I = NDR(NDT(NER(NDT(MTH(IX), 1)), 2), 3)

JNK JNK("L+1") pushes L+1, L+2, and L+3 on LAVS, updates IAC , changes the pointer of L from L+1 to L+4 and decreases the reference count of N by 1 .

BIBLIOGRAPHY

[1] "Introduction to Semi-Automated Mathematics" J. H. Bennett, W. B. Easton, J. R. Guard, and T. H. Mott, Jr. Final Report No. AFCRL 63-180. April 15, 1963. (Contract No. AF 19(628)-468)

[2] "Automated Logic for Semi-Automated Mathematics" J. R. Guard. Scientific Report No. 1 AFCRL 64-411. March 30, 1964 (Contract No. AF 19(628)-3250)

[3] "Toward Semi-Automated Mathematics: The Language and Logic of SAM III" J. H. Bennett, W. B. Easton, J. R. Guard, and T. H. Mott, Jr. Scientific Report No. 2. AFCRL 64-563. May 1, 1964. (Contract No. AF 19(628)-3250)

[4] "Semi-Automated Mathematics: SAM IV" J. H. Bennett, W. B. Easton, J. R. Guard, D. B. Loveman, T. H. Mott, Jr. Scientific Report No. 3. AFCRL 64-827. October 15, 1964. (Contract No. AF 19(628)-3250)

[5] "A Matching Procedure for ω-Order Logic" by William Eben Gould. Scientific Report No 4. AFCRL 66-781. October 15, 1966. (Contract No. AF 19(628)-3250)

[6] Robert Bumcrot, Proceedings of the Glasgow Mathematical Association, Vol. 7, Part 1, 1965, pps. 22-23

[7] "CRT-Aided Semi-Automated Mathematics" Semi-annual Report covering period: July 1, 1965 through December 31, 1965. (Contract No. AF 19 (628)-3250).

[8] "CRT-Aided Semi-Automated Mathematics" Semi-annual Report covering period: January 1, 1966 through June 30, 1966. (Contract No. AF 19(628)-3250).

Unclassified

Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Applied Logic Corporation
One Palmer Square
Princeton, New Jersey

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3. REPORT TITLE

CRT-Aided Semi-Automated Mathematics

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Final Scientific Report 1 June 1963 through 30 September 1966

5. AUTHOR(S) *(Last name, first name, initial)*

Bennett, James H.
Easton, William B.
Guard, James R.
Settle, Larry G.

6. REPORT DATE
January 1967

7a. TOTAL NO. OF PAGES
60

7b. NO. OF REFS
6

8a. CONTRACT OR GRANT NO.
AF19(628)-3250 ARPA Order No. 700

b. PROJECT NO.
8672

c. ~~XXX~~ DoD Element No. 6154501R

d. DoD Subelement n/a

9a. ORIGINATOR'S REPORT NUMBER(S)
Final Report

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
AFCRL-67-0167

10. AVAILABILITY/LIMITATION NOTICES

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

11. SUPPLEMENTARY NOTES
Prepared for
Hq., AFCRL, OAR(CRB)
United States Air Force
L. G. Hanscom Field, Bedford, Mass.

12. SPONSORING MILITARY ACTIVITY
Advanced Research Projects Agency

13. ABSTRACT

~~This~~ THE report describes the status of the fifth in a series of five experiments in semi-automated mathematics. These experiments culminated in large complex computer programs which allow a mathematician to prove mathematical theorems on a man-machine basis. SAM V, the fifth program, uses a cathode ray tube as the principal interface between the mathematician and a high speed digital computer. An elaborate language and logical capability has been implemented in SAM V. These include I/O languages for expressing mathematical statements in a form suitable for both the mathematician and the machine to recognize and handle with ease and convenience; a language for expressing and handling sorts and range of symbols; and an auto-logic algorithm and matching routine. The latter constitute the capability for handling, automatically, logic with equality. This capability is particularly useful at an intermediate state of the proof when it is desired to have the machine try and verify automatically a given portion of the proof.

Unclassified

Security Classification